A Suite of Tools for Debugging Distributed Autonomous Systems

David Kortenkamp[1], Reid Simmons[2], Tod Milam[1], and Joaquín L. Fernández[2]

[1]Metrica Inc./TRACLabs
1012 Hercules
Houston, TX USA 77058
{korten, tmilam}@traclabs.com

[2]Computer Science Dept. Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15214 USA
{reids, joaquin}@cs.cmu.edu

ABSTRACT

This paper describes a set of tools that allows a developer to instrument an autonomous control system to log data at run-time and then analyze that data to verify correct program behavior. Analysis is done using a new interval logic that allows system engineers to express complex, temporal specifications to be checked against the logged data of the autonomous control program. A feature of both the logging and analysis is that they can work with distributed programs. All data is synchronized into a common database. The data logging tools and the interval logic are fully implemented. Results are given from a NASA distributed autonomous control system application.

1. INTRODUCTION

The cumbersome process of monitoring and displaying system data, analyzing this data for anomalies and collecting the data for future analysis is typically done using application-specific code written by developers as they code their systems. Traditional software debugging tools are not designed for distributed autonomous systems, these tools often run only on single processes, and cannot integrate data across processes. The tools described in this paper allow for real-time collection, display, and analysis of data from distributed autonomous systems.

Debugging and verifying distributed control programs is notoriously difficult [Tsai 1996], yet such programs are becoming increasingly common in complex applications. Examples are spacecraft control [Muscettola et al 1998], process control [Bonasso 2001], multiple robot applications [Simmons et al 2000] and production plant control [Kresbach and Musliner 1998]. In each of these instances, concurrent programs run (often on separate machines) to generate control commands for a single or multiple devices.

The difficulty in debugging such applications is directly related to their distributed nature. When a problem occurs it can often be difficult to isolate the problem to one specific control module due to timing constraints, interprocess communication, and synchronization. The traditional, dynamic method for debugging sequential software has no timing constraints. For these systems, *cyclic debugging* (running the program until an error shows up, examining the program state, inserting assertions and re-executing the program to obtain additional information) is commonly used [Tsai et al 1996]. However, there are several reasons why this approach cannot be applied to distributed control programs:

- Often the distributed processes cannot be paused for examination since they are controlling physical hardware.
- There is no central, global state or even global clock to reference state values, which makes it difficult to reason about the "state" of the system at a given time.
- Due to latencies and timing issues, distributed control programs are inherently non-deterministic and non-repeatable.

Moreover, the questions posed by developers of distributed autonomous control systems about their systems often differ significantly from those posed of traditional, linear control programs. Analysis of cross-system data is of particular importance, including questions such as:

- Do two states in two control programs change together? What is the latency between a change in one and a change in the other?
- When event X occurs in one system, how long before event Y occurs in a second system?

This paper presents a suite of data collection tools and a real-time interval logic that is used to analyze data collected by the tools. The data collection tools and logic work together via a database to facilitate debugging and verifying distributed programs. The real-time interval logic is used to determine if the execution of a real-time distributed program, as characterized by a captured execution trace, is consistent with a formal description of the program behavior. The logic includes mechanisms to deal with metric time, powerful interval and event specification mechanisms, and different ways to deal with sets of intervals and events. We illustrate the use of our tools by applying them to validate part of an autonomous control system for NASA's Advanced Water Recovery System [Bonasso 2001].

2. PREVIOUS WORK

Stethoscope, a commercial product for collecting data from real-time programs, allows for data collection, display, and modification [Schneider 1995]. However, it is limited to real-time programs running under VxWorks and does not offer support for the kind of high-level, cross-system debugging that distributed systems require.

Tools for debugging and verifying parallel systems have recently been developed. For example, *ParaGraph* [Heath & Etheridge 1991] provides a variety of visualizations of a parallel system. Similarly, *tnfview* is a tool for debugging and verifying multi-threaded programs [Kleiman et al 1996]. None of these tools, however, offer the cross-system and high-level debugging and verification support needed for debugging autonomous systems.

As for analyzing data, a temporal logic is a good candidate for expressing specifications to verify execution trace data, since it can specify properties of event and state sequences. However, traditional linear-time temporal logic, such as PTL [Gabbay 1980] and ITL [Moszkowski 1986], or branching-time logic, such as CTL [Emerson 1982], cannot specify the quantitative aspect of time. These logics deal with concepts of eventuality, fairness, etc., which are basically qualitative treatments of time. While we can use such logics for specifications such as "*Every stimulus p is followed by a reaction q*" ($\mathbf{\Box(p \rightarrow \Diamond q)}$), it is not possible to express "*Every event p is followed by a reaction q in the next 4 time units*."

Researchers have investigated different methods to overcome this shortcoming [Tsai 1996]. One is the use of explicit clock variables, such as a global clock, that binds a variable to the corresponding time when an event occurs. In particular, this approach is used in TPTL [Alur 1990] and XCTL [Harel 1990]. Another approach, exemplified by Metric TL [Koymans 1990], is to use bounded temporal operators to restrict the time span between two events. A third approach uses time functions, as is done in RTL [Jahanian 1987].

Most of these logics were designed for model checking and they restrict their language to be able to apply verification methods. However, other logics such as Event-based Real-time Logic (ERL) [Chen 1991] and Real-time Interval Logic (RTIL) [Razouk 1989] were developed to yield practical tools for software testers running the system and checking the specifications over the trace data.

3. INTERVAL TEMPORAL CHECKING LOGIC

While temporal logics typically provide good low-level mechanisms for expressing sequencing behavior, using them to reasoning about an entire computation is often awkward and convoluted. Coupled with our desire to facilitate the expression of complex timing and relational properties of real-time distributed software, we created our own logic, based on RTL, ERL and RTIL. ITCAL (Interval Temporal Checking Logic) includes new operators to handle sets of intervals and events, introduces new structures such as *value sets*, and extends others, such as time points, already used in RTIL. From RTL we borrow operators such as universal ($\forall$) and existential ($\exists$) quantifiers, while from ERL we borrow some functionality to work with events.

Due to the focus on checking and debugging, ITCL is based on actions and system status that are defined as intervals. Reasoning about intervals allows us to easily define timing and relational properties of real-time distributed systems, such as periodic behavior or temporal constraints.

An event ω is defined as a log entry in the logged data. Log entries record relevant changes to the system, including the beginning and end of significant actions, changes to state variables, and perceived changes in the environment. The information recorded for each log entry contains the event name, a timestamp, and a set of variables associated with the kind of log entry. Events can also be defined as a log entry *type* (rather than a single log entry *instance*). An event defined in this way can have several occurrences in the trace file. We use the term "*event set*" (represented as $\mathbf{\Omega}$) to denote all of them.

We define a *time point set* ($\mathbf{\Phi}$) as a set of *time points* ($\mathbf{\phi}$) in the trace interval. A *time point set* can correspond with an *event set* or can be derived from it. For example, it is possible to define a *time point set* as the set of all the time points corresponding to the events "start_actionA" (which is an *event set*). However, we can also define a new time point set as all time points 5 seconds before the "start_actionA" events. The time point set defined this way is not an event set.

Intervals (γ) are defined as a pair of time points ($\mathbf{\phi 1.1}$ start and $\mathbf{\phi 2.1}$ end) delimiting the start and end of the interval. Usually, the two time points are events. The first time point is included in the interval and the second is not. Therefore, the starting and ending time points must be different. The whole trace itself is also considered to be an interval. As with events and time points, intervals can be grouped into *interval sets*.

Specifications to be checked against a trace file can be defined and evaluated with respect to either intervals or interval sets. The specifications consist of propositions or logical expressions defined according to ITCL.

3.1 ITCL syntax

Time points are the basic building blocks of ITCL. Time points are defined formally as:

$$\phi \equiv |\uparrow\gamma \mid \downarrow\gamma \mid \phi \rightarrow \mathbf{t} \mid \mathbf{t} \leftarrow \phi \mid \mathbf{t} \mid \omega,$$

where γ is an interval, $\mathbf{t}$ is a time value and ω is an event (log entry). The operators $\uparrow$ and $\downarrow$ appearing before an interval represent the beginning and ending of the interval. The expressions $\phi \rightarrow \mathbf{t}$ and $\mathbf{t} \leftarrow \phi$ represent the time points $\mathbf{t}$ time units after ϕ and $\mathbf{t}$ time units before ϕ respectively.

The formal definition of an interval is as follows:

$$\gamma\mathbf{:} = \phi_1 \Rightarrow \mathbf{T}_2 \Rightarrow \mathbf{T}_1 \ldots \mid \ldots \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \Leftarrow \phi_1 \mid \perp,$$

where $\mathbf{T}$ is a time point ϕ or a time point set $\mathbf{\Phi}$. The search operators ($\Rightarrow$,$\Leftarrow$) extend the interval from a starting time point searching forward ($\Rightarrow$) or backward ($\Leftarrow$) to an ending time point. Multiple search operators can be included in the same interval definition, but they all must have the same direction. If no time point is specified, searching starts from the beginning ($\Rightarrow \Phi_2$) or end ($\phi_{2.1} \Leftarrow$) of the logged data. Thus, $\Rightarrow$ represents the interval including the whole execution trace. $\perp$ is used to represent the null interval, i.e., there is no interval for which the definition holds. Since the search operators ($\Rightarrow$, $\Leftarrow$) always start searching right after the starting time point, intervals defined this way always have duration greater than zero.

Time point sets have a similar definition to time points and are defined as:

$$\Phi \equiv \Omega | \uparrow\Gamma \mid \downarrow\Gamma \mid \Phi\rightarrow t \mid t\leftarrow\Phi \mid [x\in \Phi \text{ st } P],$$

where **Ω** is an event set as defined above, and the operators ↑ and ↓ appearing before an interval set (Γ) represent the beginning and ending time points of the intervals that belong to Γ. The expression **[x∈ Φ st P]** denotes a new time point set that includes all the time points from the set **Φ** that are consistent with the condition **P**. The condition that can be used depends on the type of time point set. For example, if the time point set is a value set, **P** can be a formula that imposes a restriction on the values.

The formal definition of an interval set is as follows:

$$\Gamma: = [\gamma] \mid [P] \mid \Phi_1 \Rightarrow T_2 \Rightarrow T_5 \Rightarrow \ldots \mid \ldots \Leftarrow T_1 \Leftarrow \Phi_2 \mid [x\in\Gamma \text{ st } P] \mid I \cup I \mid I \cap I \mid I \,\&\, I \mid I \mid I$$

where **P** is a condition or logical expression, **T** is a time point ϕ or a time point set Φ, and **I** is an interval γ or an interval set Γ. An interval in brackets (**[γ]**) is an operator that converts the interval γ into an interval set that contains only one interval (γ). The next section presents some examples of how to define interval sets from a condition **[P]**, using the search operators (**⇒**, ⇐) and *conditional interval sets* (**[x∈Γ st P]**). Some examples are shown in Figure 1.

ITCL includes operators that combine intervals and interval sets in various ways. Figure 2 shows some examples of these operators, including **union** (∪), **subtraction** (∩), **disjunction** (|) and **logical and** (**&**). Their semantics are described in Section 3.2.

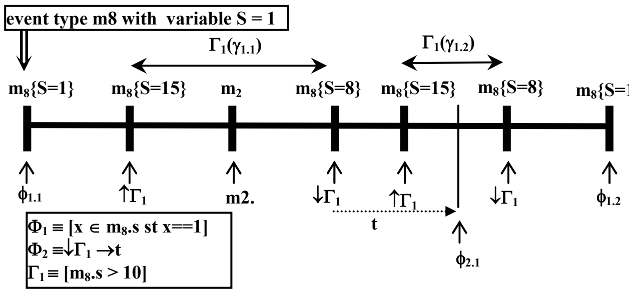
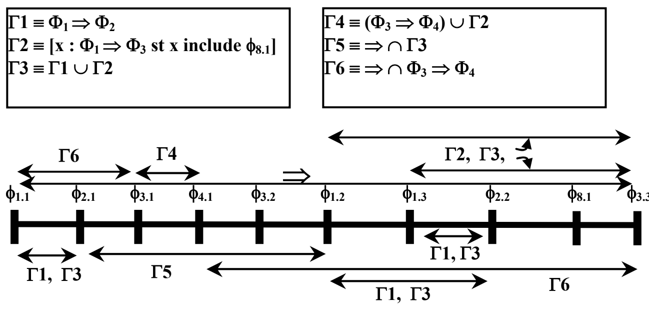


Figure 1. Time point sets and interval sets.



$\Gamma1 \equiv \Phi_1 \Rightarrow \Phi_2$
$\Gamma2 \equiv [x : \Phi_1 \Rightarrow \Phi_3 \text{ st x include } \phi_{8.1}]$
$\Gamma3 \equiv \Gamma1 \cup \Gamma2$
$\Gamma4 \equiv (\Phi_3 \Rightarrow \Phi_4) \cup \Gamma2$
$\Gamma5 \equiv \Rightarrow \cap \Gamma3$
$\Gamma6 \equiv \Rightarrow \cap \Phi_3 \Rightarrow \Phi_4$
Γ6
Γ4
⇒
Γ2, Γ3,
$\phi_{1.1}$
$\phi_{2.1}$
$\phi_{3.1}$
$\phi_{4.1}$
$\phi_{3.2}$
$\phi_{1.2}$
$\phi_{1.3}$
$\phi_{2.2}$
$\phi_{8.1}$
$\phi_{3.3}$
Γ1, Γ3
Γ5
Γ1, Γ3
Γ1, Γ3
Γ6

Figure 2. Operations with intervals.

Specifications are formed from logical expressions (**P**) joined by the Boolean operators **and** (∧), **or** (∨) and **implies** (**→**). These expressions can also have the **not** (¬) operator preceding them. Logical expressions (**P**) are either relational expressions or temporal relations. Universal (∀) and existential (∃) quantifiers can be used to evaluate an expression over a set of items (events, intervals or values).

Relational expressions (<, ≤, >, ≥, =, ≠) can also be used with a value set. The result is an interval set consisting of the intervals where the expression holds true. For example, in Figure 1, the expression [$m_8.s>10$] returns an interval set that represents all occurrences of message m_8 where the ***S*** field has a value greater than 10.

ITCL contains several time-related operators. The **time** operator returns the timestamp of an event. The **duration** operator returns either the duration of an interval or a value set containing the durations of all intervals within an interval set. The temporal operator **always** (γ ⊗ **P**) is true if **P** is true during all the minimal intervals within γ. ITCL also defines the **eventually** operator (γ ◊ **P**), which is equivalent to **¬(**γ **⊗¬P)**. Table 1 shows the more commonly used operators. Note that ITCL also includes arithmetic (+, -, *, /), logical (¬, ∧, ∨), and relational (=, ≠, <, ≤, >, ≥) operators.

Table 1. Symbol equivalence

ITCL	Short description
∀	**For all**
∃	**Exists**
∈	**Belongs to**
⇒	**Search forward**
⇐	**Search backward**
→	**Extend forward**
←	**Extend backward**
∪	**Union of two interval sets**
∩	**Subtraction of interval sets**
↑	**Beginning an interval**
↓	**Ending an interval**
Δ	**Evaluate at the beginning**
∇	**Evaluate at the end**
β	**Evaluate after**
α	**Evaluate before**
⊗	**Always**
◊	**Eventually**
⊥	**Null interval**
st	**Restricts**
time ϕ	**Timestamp of time point ϕ**
|Γ/Φ|	**Cardinality of the set**
Miscellaneous symbols	
≡	**Assignation**
=>	**implies: {a=>b} ↔ {!a ||b}**
Print	**Print item or set of items**
MAX_TRACE	**Timestamp of last event**
MIN_TRACE	**Timestamp of first event**
Maxvalue(v)	**Maximum value of value set v**
Minvalue(v)	**Minimum value of value set v**

3.2 *Writing Specifications using ITCL*

The main design goal for ITCL is to provide a general and flexible language with which to specify the execution of autonomous systems. ITCL is well suited for this. Execution is typically characterized by the occurrence of events, changes to state variables, and continuity of values over time. ITCL's focus on sets of events, values, and intervals maps well to this.

Typically, however, a user is confronted with different ways to specify desired actions and states that depend on the information available in the log data. For example, consider the task of specifying all the intervals during which the robot performs a "rotate" action. If we log the events when the robot starts rotating (**start_rot**) and stops rotating (**end_rot**), we can use the search operator to define "rotate":

$$\textbf{rotate} = \textbf{start\_rot} \Rightarrow \textbf{end\_rot}.$$

However, if the log data contains information about when the rotational speed of the robot (**rot_speed**) changes (the event **change_val**), the same "rotate" action can be defined as:

$$\textbf{rotate} = [\textbf{change\_val.rot\_speed} > \varepsilon],$$

where ε can be either zero or a threshold over which we consider the robot to be rotating. Note that, in either case, the result is an interval set, which corresponds to the idea that the robot could execute the "rotate" action many times during a single run of the system.

Operations that combine interval sets ($\mathbf{I_1 \cup I_2}$, $\mathbf{I_1 \cap I_2}$, $\mathbf{I_1\ \&\ I_2}$, $\mathbf{I_1 \mid I_2}$) can be used to succinctly specify more complex types of constraints. For example, to specify "condition P must hold after executing action A1, A2, or A3" we can use the **union** and **evaluate after** operators:

$$\mathbf{(A1 \cup A2 \cup A3)\ \beta\ P,}$$

this works because the **union** operator creates a new interval set that includes *all* the intervals of its arguments.

Similarly, we often want to specify that some condition will never occur unless the system is in a particular state. For example, we might want to specify that "P must never hold unless the system is executing action A1." In this case, we can use the **subtraction** operator to find all the intervals where A1 is *not* occurring, and specify that P should never hold during those intervals:

$$\mathbf{([\Rightarrow] \cap A1) \otimes \neg P}$$

While ITCL is very general, sometimes the specifications are not very readable. Based on our experience with ITCL, we are adding higher-level constructs ("syntactic sugar") to make it easier to specify expressions that appear frequently. Table 2 shows some of the extensions, together with their expansions, in ITCL.

4. DATA COLLECTION

The data collection demands of distributed control programs range from low-level sensory data to the programs' internal states. The data collection routines have the following requirements:

Table 2. ICTL extensions to allow easier use

MACRO	EQUIVALENCE
$\gamma 1$ intersects $\gamma 2$	$\mathbf{time(\uparrow\gamma1) < time(\downarrow\gamma2) \wedge}$ $\mathbf{time(\downarrow\gamma1) > time(\uparrow\gamma2)}$
$\gamma 1$ include ϕ	$\mathbf{time(\uparrow\gamma1) \le time(\phi) \wedge}$ $\mathbf{time(\downarrow\gamma1) > time(\phi)}$
$\gamma 1$ include $\gamma 2$	$\mathbf{time(\uparrow\gamma1) \le time(\uparrow\gamma2) \wedge}$ $\mathbf{time(\downarrow\gamma1) \ge time(\downarrow\gamma2)}$
$\gamma 1$ inside $\gamma 2$	$\mathbf{time(\uparrow\gamma1) \ge time(\uparrow\gamma2) \wedge}$ $\mathbf{time(\downarrow\gamma1) \le time(\downarrow\gamma2)}$
ϕ inside $\gamma 2$	$\mathbf{time(\uparrow\gamma2) \le time(\phi) \wedge}$ $\mathbf{time(\downarrow\gamma2) > time(\phi)}$
ϕ_1 isbefore [T1, T2] ϕ_2	$\mathbf{time(\phi_1) + T1 \le time(\phi_2) \wedge}$ $\mathbf{time(\phi_1) + T2 \ge time(\phi_2)}$
ϕ_1 isbefore (T1, T2] ϕ_2	$\mathbf{time(\phi_1) + T1 < time(\phi_2) \wedge}$ $\mathbf{time(\phi_1) + T2 \ge time(\phi_2)}$
ϕ_1 isbefore [T1, T2) ϕ_2	$\mathbf{time(\phi_1) + T1 \le time(\phi_2) \wedge}$ $\mathbf{time(\phi_1) + T2 > time(\phi_2)}$
ϕ_1 isbefore (T1, T2) ϕ_2	$\mathbf{time(\phi_1) + T1 < time(\phi_2) \wedge}$ $\mathbf{time(\phi_1) + T2 > time(\phi_2)}$
closeto(v, co, ε)	$\mathbf{v < (co + \varepsilon) \wedge v > (co - \varepsilon)}$

- Data collection in real time
- Data logging to a database
- Grouping of data into logical sets
- Triggering options (e.g., allowing only certain data in certain ranges to be collected)
- Change-only logging

Our goal for data collection is to replicate the ease-of-use of the *printf* command in C, while allowing for more control and for distributed operation. In essence, what we have implemented is a *remote printf* capability named *rlog*.

Rlog is a set of libraries that allows users to easily instrument their programs and send the output to a variety of destinations, such as the screen, a file, a remote computer, or a database. The types of data that can be logged are similar to that of *printf*: character, unsigned character, short integer, unsigned short, integer, unsigned integer, long integer, unsigned long, floating point, double floating point, and character string.

We have implemented a variety of logging functions, ranging from logging a single variable, to logging multiple variables at once, to conditional and change-only logging. In addition, *rlog* includes a pre-processor that enables function entries and exits to be logged automatically. Logged values can be directed at run-time to a variety of output destinations, including the screen, a file, a remote computer, or an SQL relational database. It is the database feature that allows for multiple processes to be logged to a single location. All of the analysis tools described in Section 3 get their data from the database. Figure 3 shows the general system set-up.

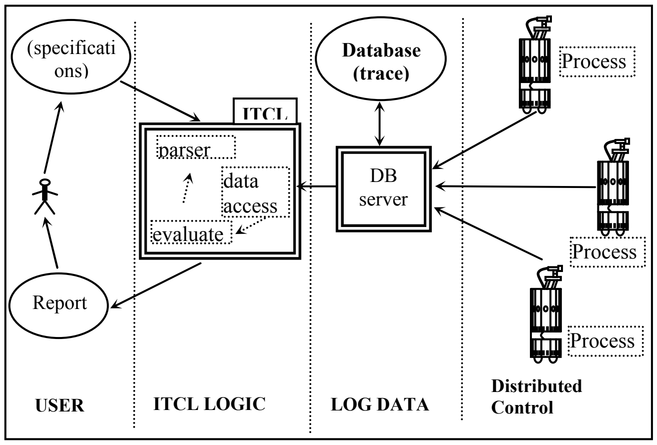


Figure 3. General architecture.

Distributed logging to a central database works as follows: A socket connection is made between each process being logged and a central data collection process. The central data collection process communicates with the other processes to determine clock offsets for each of them. It then uses this clock offset to synchronize all the event times for each logged datum. Once times are synchronized, the data is entered into the relational database.

We have collected performance data on our logging tools. The data was collected on an 800Mhz Intel Pentium III running RedHat Linux 6.2. Table 3 shows the number of seconds it took to call the basic *rlog* function 100 times for different output possibilities. The results are an average for all the different data types that can be logged, plus any associated initialization and clean up. Note that the first line of the table (NULL) is where the logging functions are invoked, but do not actually output any data.

Null	0.009
File	0.053
Screen	0.534
TCP/IP socket	0.711
SQL database	0.347

Table 3. Time in seconds for invoking the logging functions 100 times for different output locations

5. EXPERIMENTS

We validated our approach by collecting data from a test of the NASA Advanced Water Recovery System (AWRS). The AWRS is an automated system being developed to support astronauts on very long duration missions, by recycling all water [Bonasso 2001]. Our interest was to verify that the control system was actually performing according to specifications.

We first gathered several day's worth of data from the control system using the data collection tools described in Section 4. This data was logged to a database. It was then analyzed using rules written in ITCL. This section details the results of those experiments, together with some examples of the specifications used and the reports obtained.

For this test, we used an event (**ChangedValue**) to report whenever some variable of interest in the controller process changed. Most of the variables correspond to sensors that report temperature, dew point, flow, etc.

Based on this data, we translated specifications that were provided (in English) by the WRS engineers. One such constraint is the following:

"Whenever BlowerPower is greater than 0 then FlowMeter07 must also be greater than 0"

We translate this into ITCL as follows:

[ChangedValue.BlowerPower > 0] ⊗ (ChangedValue.FlowMeter07 > 0) [1]

It is important to note that this is not the only way to express this notion. We can also use:

∀ itvl ∈[ChangedValue.BlowerPower > 0] (ChangedValue.FlowMeter07 > 0)

Expressions that include fixed time, such as periodic behaviors or events that occur before or after some action, result in very simple ITCL expressions using the start (↑), end (↓) and extending-in-time (←, →) operators. One of the most complex timing situations is that a sequential set of events should happen when the system is under some specific conditions. Once again, the WRS engineers provided us with the following English specification:

"When Switch3State and Switch1State both change to 1 then the following should happen in this order:
1. FlowMeter08 should go to a little over 7.
2. FlowMeter07 should go to a little over 7.
3. Thermocouple29 should go above 100.
It is possible for these to occur roughly simultaneously, but they should never occur in a different order."

To express this constraint in ITCL, we first define some auxiliary variables:

s1s3on ≡ [(ChangedValue.Switch3State = 1) ∧ (ChangedValue.Switch1State = 1)];
sfm8overThresh ≡ ↑([ChangedValue.FlowMeter07 >7]);
sfm7overThresh ≡ ↑ ([ChangedValue.FlowMeter08 > 0.7]);
stm29overThresh ≡ ↑([ChangedValue.Thermocouple29 >100])

Then, we define the restrictions:

```
∀it2_1 ∈ s1s3on {
 ∃ ev1_2 ∈ sfm7overThresh { it2_1 include ev1_2 ∧
   ∃ ev1_3 ∈ sfm8overThresh { it2_1 include ev1_3 ∧
     ev1_2 is_before[,] ev1_3 ∧
      ∃ ev1_4 ∈ stm29overThresh { it2_1 include ev1_4 ∧
        ev1_3 is_before[,] ev1_4 }}}}
```

[1] To make it easier for people not familiar with temporal logic (and also easier to type), the actual syntax used is: **"during [ChangedValue.BlowerPower > 0] always (ChangedValue.FlowMeter07 >0)"**

Each specification is evaluated against the logged data in the database. If the specification is found to be false, then a counterexample is generated. In particular, the system shows the first time interval in which the specification becomes false, and the reasons why. For example, our system produces the following output:

```
-- Specification:
∀ inc ∈ increasing {
 Thermocouple11(time(↑ (inc)))<Thermocouple11(time(↓(inc)))
};
 is FALSE because:
When interval inc has the value: Intervalvar=
                              Start: sec = 996622420 usec = 367780
                              End: sec = 996622421 usec = 377797
 the condition (∀) becomes false.
Operation '<' is FALSE because the operands are:
 First Operand: Longvar= 25
Second Operand: Longvar= 24
```

where the operands are the values of the Thermocouple. This report allows the engineer to know where a specification was violated and to find a solution to the problem.

6. CONCLUSIONS

Taken together, the data collection and analysis tools offer developers of distributed control programs the ability to see what their programs are doing and verify their correct behavior. Of critical importance is the usability of the tool suite – if the tools are not easy to use then developers will not adopt them. We have tried to make our logging library as easy to use as *printf*. ITCL requires more of a learning curve, but we plan to provide more "syntactic sugar" and graphical interfaces to make it easier to use. We encourage anyone interested to download our logging tools at: http://www.traclabs.com/rlog and give us feedback on how they can be improved.

7. ACKNOWLEDGEMENTS

This work is supported by NASA grant NAS2-99020 administered by NASA Ames Research Center. Pete Bonasso of Metrica Inc./TRACLabs is the chief software engineer of the WRS control system and worked with us to instrument and analyze the control code. Mark Shirley of NASA Ames Research Center was a key participant in the formulation of this project and contributed to its preliminary design.

REFERENCES

[Alur 1990] **R. Alur and T. Henzinger.** *Real-time logic: Complexity and expressiveness*. **In Proc. of IEEE 5th Symp. on logic in Computer Science, Philadelphia, pp. 401-413, June 1990**

[Bonasso 2001] **R. Peter Bonasso, "Intelligent Control of a NASA Advanced Water Recovery System," in Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS) 2001.**

[Emerson 1982] **A. E. Emerson and E. M. Clarke, *Using branching time logic to synthesize synchronization skeletons*. Science of Computer Programming, 1982.**

[Gabbay 1980] **D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. *On the temporal analysis of fairness*. In Proc. ff the 7th Annual Symposium on Principles of Programming Languages, 1980.**

[Harel 1990] **D. Harel, H. Lachover, A. Naamad, A. Pnueli.** *Explicit clock temporal logic*. **In Proc. of 5th Annual IEEE Sump. on Logic in Computer Science, Philadelphia, pp. 401-413, June 1990.**

[Heath 1991] **M. Heath and J. Etheridge, "***Visualizing the Performance of Parallel Programs***," IEEE Software 8, 1991.**

[Jahanian 1986] **Jahanian, F. and A. K. Mok, *Safety analysis of timing properties in real-time systems*. IEEE Transactions on Software Engineering 12 (1986)**

[Jahanian 1987] **Jahanian, F. and A. K. Mok, *A graph-theorem approach for timing analysis and its implementation*. IEEE Transactions on Computers, C-36(8):961-975, August 1987**

[Kleiman 1996] **S. Kleiman, D. Shah and B. Smaalders, *Programming with Threads*, SunSoft Press, Mountain View CA, 1996.**

[Kortenkamp 2001] **David Kortenkamp, Tod Milam, Reid Simmons and Joaquín López Fernández.** *Collecting and Analyzing Data from Distributed Control Programs*. **Runtime Verification 2001 (Satellite workshop to CAV'01), Paris, France 2001.**

[Koymans 1990] **R. Koymans. Specifying real-time properties with metric temporal logic. Real-Time Systems J., 1990**

[Kresback 1998] **Kurt D. Kresback and David J. Musliner, "***Applying a Procedural and Reactive Approach to Abnormal Situations in Refinery Control***," Proceedings of the Conference on Foundations of Computer-Aided Process Operations (FOCAPO), 1998.**

[Moszkowsky 1985] **Ben Moszkowski. "A Temporal Logic for Multilevel Reasoning About Hardware", IEEE Computer 1985; 18:10-19.**

[Muscettola 1998] **Nicola Muscetttola, P. Pandurang Nayak, Barney Pell and Brian C. Williams, "***Remote Agent: To Boldly Go Where No AI System Has Gone Before***," Artificial Intelligence, 103(1), 5—47, 1998**

[Razouk 1989] **Razouk, R. R. and M. M. Gorlik, A real-time interval logic for reasoning about executions of real-time programs. SIGSOFT SE Notes 114 (1989)**

[Schneider 1987] **R. Schneider, "*Real-time data monitoring and visualization*," Technical Report White Paper, available at www.rti.com, Real-Time Innovations Inc., 1987.**

[Simmons 2000] **R. Simmons, D. Apfelbaum, D. Fox, R. P. Goldman, K. Zita Haigh, D. J. Musliner, M. Pelican, and S. Thrun. "***Coordinated Deployment of Multiple, Heterogeneous Robots",* **In Proceedings of the Conference on Intelligent Robots and Systems (IROS), Takamatsu Japan, October 2000.**

[Tsai 1996] **Tsai, J., Y. Bi, S. Yang and R. Smith, *Distributed Real-Time Systems: Monitoring, Visualization and Analysis*, Wiley & Sons, New York, 1996.**